Managing complex objects in an extensible relational DBMS

Georges Gardarin, Jean-Pierre Cheiney, Gérald Kiernan, Dominique Pastre, Hervé Stora

INRIA

UNITE DE RECHERCHE
INRIA-ROCQUENCOURT

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P.105
78153 Le Chesnay Cedex
France
Tél. (1) 39 63 55 11

Rapports de Recherche

N° 981

Programme 4

MANAGING COMPLEX OBJECTS IN AN EXTENSIBLE RELATIONAL DBMS

Georges GARDARIN
Jean-Pierre CHEINEY
Gérald KIERNAN
Dominique PASTRE
Hervé STORA

Mars 1989

RR_0981

INRIA

UNITÉ DE RECHERCHE
INRIA-ROCQUENCOURT

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P.105
78153 Le Chesnay Cedex
France
Tél.:(1) 39 63 55 11

Rapports de Recherche

N° 981

Programme 4

MANAGING COMPLEX OBJECTS IN AN EXTENSIBLE RELATIONAL DBMS

Georges GARDARIN
Jean-Pierre CHEINEY
Gérald KIERNAN
Dominique PASTRE
Hervé STORA

Mars 1989

Managing Complex Objects in an Extensible Relational DBMS

Georges GARDARIN(i), J-Pierre CHEINEY(ii), Gerald KIERNAN(i),
Dominique PASTRE(iii), Hervé STORA(iv)

Abstract

This paper presents the management of complex objects in an extended relational database management system. Complex objects are built using the list constructor. Complex object types are defined as relation domains encapsulated within methods written in LISP or C. For a given application, the database administrator introduces the complex domains with the encapsulating methods within the DBMS meta-base using a specific command. User-defined domains may be described using a "IS-A" hierarchy, which makes possible the inheritance of methods among domains. The domain methods are directly used in the external data manipulation language, which is an extended version of SQL with object oriented features (i.e., structured complex objects, methods and inheritance). When executing a query, the system selects the correct methods to apply on complex objects according to the parameter types given in method calls. With the specific LISP interpreter embedded in the DBMS, errors in functions on complex objects are detected at run time. Thus, the DBMS is protected from errors arising in user programs. Methods can also be programmed in C code and compiled into object code. C object code is dynamically loaded when C programs are referenced in queries. At the internal level, an appropriate clustering strategy is used to store complex objects on disk. Relations can be clustered according to the result of user-defined methods applied to basic or user-defined domains. At the external level, graphical interface procedures are available to input and display non-standard data. A fully running version of the system is commercially available for SUN UNIX workstations.

(i) Institut National de la Recherche en Informatique et Automatique
(ii) Ecole Nationale Supérieure des Télécommunications
(iii) Eurosoft
(iv) Infosys

Gestion d'Objets Complexes dans un SGBD Relationnel Etendu

Georges GARDARIN, J-Pierre CHEINEY, Gerald KIERNAN,
Dominique PASTRE, Hervé STORA

Résumé

L'introduction dans les SGBD du traitement d'objets complexes nécessite des efforts de recherche pour définir des modèles et des langages permettant une représentation et une manipulation aisées et efficaces. Stockage et méthodes de placement adaptées doivent également être abordée. Afin d'utiliser le SGBD relationnel SABRINA dans le cadre d'applications non-classiques (CAO, Bureautique, Cartographie, Textuelles), nous présentons dans cet article une extension basée sur l'adjonction, au coeur du système, d'un interpréteur Lisp réduit. Les objets complexes sont introduits dans SABRINA grâce à la notion de domaines complexes. Ceux-ci sont constitués d'expressions Lisp et manipulés par des fonctions associées. L'administrateur de la base de données peut définir de nouveaux domaines complexes et l'ensemble des fonctions applicables à chaque domaine ou à ses spécialisations. L'usager peut alors utiliser les fonctions dans les requêtes du langage externe. Une méthode de stockage adaptée aux objets complexes est également présentée. Le placement physique est déterminé par des ensembles de prédicats qui définissent un partitionnement récursif de la relation. Ces prédicats sont exprimés par des fonctions Lisp choisies parmi celles définies sur le domaine. L'interpréteur Lisp est simplifié et optimisé par la prise en compte des particularités de son utilisation dans le SGBD. L'implantation de cette extension est présentée: le langage externe, l'architecture, l'optimisation du placement et du traitement Lisp. Elle ne nécessite que peu de modifications du SGBD, présente une puissance d'expression importante et permet l'efficacité des accès. Cette implantation est actuellement opérationnelle.

Cette recherche a été financée par le CNET Convention No.887B035007909245LASLCLSR, No.867B078007904245LASLCLSR et le PRC-BD3

Managing Complex Objects in an Extensible Relational DBMS

Georges GARDARIN(i), J-Pierre CHEINEY(ii), Gerald KIERNAN(i),
Dominique PASTRE(iii), Hervé STORA(iv)

Abstract

This paper presents the management of complex objects in an extended relational database management system. Complex objects are built using the list constructor. Complex object types are defined as relation domains encapsulated within methods written in LISP or C. For a given application, the database administrator introduces the complex domains with the encapsulating methods within the DBMS meta-base using a specific command. User-defined domains may be described using a "IS-A" hierarchy, which makes possible the inheritance of methods among domains. The domain methods are directly used in the external data manipulation language, which is an extended version of SQL with object oriented features (i.e., structured complex objects, methods and inheritance). When executing a query, the system selects the correct methods to apply on complex objects according to the parameter types given in method calls. With the specific LISP interpreter embedded in the DBMS, errors in functions on complex objects are detected at run time. Thus, the DBMS is protected from errors arising in user programs. Methods can also be programmed in C code and compiled into object code. C object code is dynamically loaded when C programs are referenced in queries. At the internal level, an appropriate clustering strategy is used to store complex objects on disk. Relations can be clustered according to the result of user-defined methods applied to basic or user-defined domains. At the external level, graphical interface procedures are available to input and display non-standard data. A fully running version of the system is commercially available for SUN UNIX workstations.

(i) Institut National de la Recherche en Informatique et Automatique
(ii) Ecole Nationale Supérieure des Télécommunications
(iii) Eurosoft
(iv) Infosys

1. Introduction

The appeal of relational database systems for standard applications is due to the simplicity of the relational model. However, relational systems fail to cater to new applications such as office, CAD/CAM, CASE or geographical applications, which are characterized by complex data types and operations. For such applications, relational database systems may serve as storage systems on which are built the application software. A first approach to extend relational systems consists in adding an interface layer over an existing relational database system. The interface simulates the extended model by converting schemas and queries into their relational counterpart. This is the approach used in GEM [Tsur84], [Zaniolo83,85] which offers an entity-relationship database interface. The attractiveness of such an approach lies in its inexpensive implementation using

reliable existing technology. However, the shortfall is performances. The greater the difference between the end-user model and the database model, the more complex is the translation process leading to eventual inefficiency.

Recent publications point out the need to include user-defined data types within the relational systems [Gardarin89]. User-defined data types allow the users of relational systems to tailor the database system to the needs of their specific applications [Ong84], [Osborn86], [Stonbraker83,86]. User-defined domains are operationaly defined, that is, the semantics of new data-types are the operations which can be performed over them. The programs which manipulate these new data types are registered within the DBMS system and dynamically linked at DBMS run-time. Other attempts to extend relational systems have been done using non first normal form data models. Complex objects retain more of their semantics by using richer data models which permit to capture for example the hierarchical quality of data [Bancilhon86], [Schek86], [Verso86], [Zaniolo85].

The approach proposed in this paper extends the notion of domain to support complex objects. In most implementations of relational systems, available domains are limited to integers, real number and character strings. Certain extensions have added date and money. In Codd's relational model [Codd70,79], the notion of domain is defined by a set of values. No restriction is made on the types of values which can be represented as domains. Extending relational domains to include user-defined domains has been initially presented in [Stonebraker83,86], [Ong84], for the INGRES database system and in [Osborn86] for the RAD database system. In both these systems, the relational model is extended to include user-defined domains with their associated methods. The external query languages of these systems are extended to include user-defined methods within relational expressions. User-defined methods can appear in any clause standard operators appear (projection, restriction, ...). The user programs methods in a programming language which can be compiled or interpreted. Other aspects of the database system have also to be reconsidered. Efficiency is maintained by clustering relations with frequently applied methods [Cheiney88], [Wilms88] and also by cost evaluations for query optimization [Schwartz86], [Carey86].

Different implementations of User-defined Data Types (UDT) can vary according to the following points :

(i) The possibility of defining new functions only on user-defined domains or both on UDT and on basic domains;
(ii) The choice of programming language in which to implement user-defined operations;
(iii) The possibility to use existing code to implement new operations;
(iv) The existence of an "IS-A" hierarchy among domains to establish operation inheritance;
(v) The execution of operations : dynamically linked or interpreted;
(vi) The presence of clustering methods for UDT.

Based on these criteria, our approach :

(i) allows to define new methods (i.e., functions attached to objects) on basic or complex domains;

(ii) uses an object oriented version of LISP for UDT programming;

(iii) allows to reuse any method or code which is registered within the DBMS to build new UDT;

(iv) defines UDT using an "IS-A" hierarchy to apply method inheritance among domains;

(v) is based on a special implementation of a LISP interpreter. Although, UDT operations can be programmed in C source code and compiled into object code which is linked dynamically at run-time. LISP code can be interfaced with the C code. Errors arising in C programs are not as easily managed as those in LISP.

(vi) allows a complex object clustering method based on predicate trees.

Using these principles, we extend an existing relational DBMS named SABRINA, developed at INRIA in the beginning of the 80s [Gardarin87]. In the extended system, a UDT is defined as a method which is used to create and validate the occurrences of the domain, and methods that are applicable to this domain. The name of the method which is used to create and validate the occurrences of the domain is also used as the name of the domain. The declaration of a generalization hierarchy for each complex domain allows dynamic inheritance; a method is selected according to its parameters types. This provides greater flexibility and consistency than having each method attached to a unique domain type. When methods are programmed in LISP, the run-time is interpreted; this allows managing programming errors. However, it is also possible to dynamically link and run C object code. This feature provides better performances than interpreting LISP code. However, if no C code is used, no external files need to be managed; the methods codes and relative information are stored within the database system and hold in one relation. To maintain performances with LISP, an appropriate design of LISP has been implemented. This design differs from standard LISP implementations in error handling and garbage collection. No global environment variables are permitted thus simplifying and optimizing the garbage collection process.

This article presents the manipulation of complex objects in an extended relational DBMS. Complex objects are integrated as user-defined data types (UDT) at the domain level, which makes the system extensible. The extensible system is currently available as a product (SABRINA, version 7). Apart from this introduction, section 2 details the definition of UDTs using an object oriented version of the LISP language based on a subset of the LeLisp syntax [Chailloux86]. Section 3 describes the extensions to the external SQL interface for manipulating UDTs. The overall system architecture is described in section 4 with special emphasis on the alterations brought to the system for UDT manipulation. The next section describes the partitioning of relations on disk for optimal retrieval of relations containing UDT. This is done using a clustering strategy based on predicate trees and partitioning relations on the results of frequently applied methods. Section 6 deals with further strategies for optimizing query processing in the UDT

framework. Section 7 discusses special interface methods for managing graphical information. The conclusion terminates the paper.

2. User-defined Data Types: Concepts and Language

2.1. User Data Types and Methods

The basic domain types commonly available in commercial relational systems (integers, real numbers and character strings) are insufficient to represent the types of data manipulated by new and diverse applications (graphical data types, lists, etc.). Each data type should be manipulated with a set of appropriate methods which may or may not be limited in number. Because of the diversity of types required by the new applications, it is insufficient to extend the database system to include a specific and limited set of data types. For this reason, the object oriented approach allows the user to define methods on objects in an incremental way. Our approach is similar in the sense that it lets the user define the data types and methods which are specific to the application area. Geographical applications may require geometric data types such as polygons. These can be built from simpler types such as Cartesian coordinates (points). A polygon can be represented as a list of points. Geographical districts can then be described as a specific form (or specialization) of polygon. Specialized methods for geographical districts can then be defined, for example, the surface method returns in acres the surface area of a district. If districts are described as a specialization of polygons, then all methods applicable to polygons are also applicable to districts.

While the implementation remains altogether relational, this extends the relational system to cater to a wider variety of application areas. It also integrate in the relational system a few concepts of the object oriented approach, namely complex objects, methods, inheritance and extensibility. The LISP language was chosen as a basis to build a UDT programming environment for the following reasons : i) LISP is a powerful language for manipulating complex structured objects built with the list constructor; structures such as lists, sets and trees are easily manipulated in LISP; because LISP is a weakly typed language, UDT could be defined operationaly instead of structuraly; ii) the functional approach seemed appropriate for UDT methods; new methods can be built from existing ones; iii) domain hierarchies and inheritance can be handled by LISP; iv) user programming errors could be easily managed by specific controls over basic LISP functions; v) the programming environment could be completely integrated within the database environment having function code retrieved from the database selectively; vi) specialized LISP functions could be easily added to manipulate UDT; for example, for storing and modifying UDT contained in the database. The LISP interpreter is an entry point from which UDT are updated and registered within the database system [Kiernan87].

2.2. Complex Domain Definition

To create a new UDT, the user writes an initial method that creates and validates instances of the new domain. This method will be used automatically by the integrity mechanisms either when instances are inserted or modified. The name of this method is used as the name of the domain. Once a domain name is defined, additional methods can be declared with references to these domains as parameter types. Each method parameter must be typed using a basic domain (integer, real or text) or a complex domain declared in the inheritance hierarchy of complex domains whose root is denoted # (complex object). In this way, method inheritance can be applied. For example, the surface method can be inherited from rectangle to square. The complex domain hierarchy is represented using the notion of package found in the LeLisp language [Chailloux86]. Thus, a domain is a path name from the root (labeled as #) to a terminal name in the hierarchy. For example, the square domain is described as a specialization of rectangles as follows : #:rectangle:square.

Hence, registering a new domain requires defining the domain integrity in the form of a validation method and specifying the place of the domain in the generalization hierarchy. A UDT is described as a LISP function and comprises the domain name, the generic domain names, and the validation function body definition. All methods applicable to a generic domain are also applicable to the domain itself unless it has been redefined at that level. The dd function is used to define new UDT. The syntax of the dd function is as follows :

(dd <domain name> (<parameter>) <function body>)

For example, the following function defines the RECTANGLE domain :

```
(dd #:RECTANGLE (x)
 (and (listp x)
 (numberp (car x))
 (numberp (car (cdr x)))
 (null (cdr (cdr x)))))
```

According to this function, for an object x to be a rectangle, it has to be a list of two elements where both elements are numbers. The first element is the height of the rectangle, the second is the width.

2.3. Method Definition

A UDT method is defined also as a LISP function. The de and the df functions are used to define UDT methods. These two functions are standard in current implementations of LISP. The

general syntax of these functions is :

(de <method name> (<parameter list>) <method body>)

The method name is the name of the UDT method being defined. The list of parameters are the arguments to which methods values will be bound to at method run-time. The method body implements the method. Other UDT can be referenced in the method body. The method name is composed of a name which is preceded by a list of domain types, one per parameter in the parameter list; thus identifying each parameter's type. For example, the LENGTH method which is applicable to RECTANGLES will be named #:(#:RECTANGLE):LENGTH. This notation insures that all parameter types are taken into consideration when selecting a method. Hence, methods may be overloaded according to all parameter types.

The following methods are defined for RECTANGLES:

```
(de #:(#:RECTANGLE):HEIGHT (x) (car x))

(de #:(#:RECTANGLE):WIDTH (x) (car (cdr x)))

(de #:(#:RECTANGLE):SURFACE (x) (* (:width x)(:height x)))

(de #:(#:RECTANGLE #:RECTANGLE):HIGHEST (x y)
  (if (> (:height x)(:height y)) x y))
```

The HEIGHT method extracts the first number in the two element list as the height of the rectangle. The WIDTH method does the same for the second element. The SURFACE method multiplies the height by the width to obtain the surface. The HIGHEST method returns the rectangle with the greatest height. Methods can also be programmed in C source code, compiled and dynamically linked with the DBMS program at run-time. For example, the surface method may defined as follows :

```
(dc #:(#:RECTANGLE):SURFACE (x) "/usr/mydir/surface.o")
```

The dc function tells the interpreter that the SURFACE method is programmed in C. LISP and C function can be mixed. Programming errors occurring in C functions may cause the DBMS program to terminate abnormally while error occurring in LISP functions are controlled by the interpreter. Although C code runs faster than LISP code, there is a non negligible amount of time required by the loader to load and link object code to the DBMS run-time.

To define new UDTs, the user obtains the general UDT programming environment from within the external SQL language by typing the LISP command. An example of a UDT programming session follows. In this session, a new UDT method named BIGGEST is defined for RECTANGLES. The BIGGEST method accepts two arguments of type RECTANGLE and returns as result, the RECTANGLE with the greatest surface value. Once the method is defined, it is tested on trial data and then registered within the DBMS system. The SAVE function serves this purpose. The SAVE function takes two arguments: the name of the UDT and the type of result returned by the UDT. The result type is needed by the DBMS to manage coherency in relational expressions. END terminates the session and control is returned to SQL. At which point the user can include the BIGGEST method in appropriate relational expressions.

```
> LISP ;

? (de #:(#:RECTANGLE #:RECTANGLE):BIGGEST (x y)
?  (if (> (:SURFACE x) (:SURFACE y)) x y))

= #:(#:RECTANGLE #:RECTANGLE):BIGGEST

? (:BIGGEST '(4 5) '(2 3))
= (4 5)

? (save #:(#:RECTANGLE #:RECTANGLE):BIGGEST #:RECTANGLE)
= #:(#:RECTANGLE #:RECTANGLE):BIGGEST

? END
[]
```

3. OBJECT-SQL : The External Language Interface

3.1. Complex Object Creation

The relational DBMS implements an SQL interface which is based on the SQL norm. The language has been extended to manipulate UDT. OBJECT-SQL is the name of this extended language interface. The database administrator extends the DBMS by defining new UDT which are then made available for use in the external language. In this section, the various extensions brought to SQL to include UDT are considered.

Relations are created using the CREATE TABLE command. For example, the RECTANGLES relation is created in the following:

```
CREATE TABLE RECTANGLES (
        R#          integer,
        COLOR       text,
        SIDES       rectangle) ;
```

This relation contains three attributes where the first two are of standard domains and the last one is a complex domain. The SIDES attribute takes its values from the RECTANGLE domain which has been defined as a list of two numbers. Once the rectangles relation has been created, values may be inserted into the relation using the INSERT command. For example,

```
INSERT INTO RECTANGLES VALUES (1, BLEU, (4 5)) ;
```

When new values are inserted into relations, the UDT method which implements domain integrity constraints validation are run over the new values to determine if the values qualify as occurrences of the domain. The same check applies when UDT values are updated. Here, the value (4 5) qualifies as an occurrence of the rectangle domain. Note that the system does not implement object identity. However, it could be possible to identify complex objects within the validation function; that would make possible referential sharing among complex objects (i.e., complex domain values). This is left for further research.

3.2. Complex Object Selection

A UDT method can be used in any clause of a relational expression (projection, restriction, aggregation, sort) and is applicable to one or more attributes. A UDT method F applied to a number n of arguments is written as F(P1..Pn). The parameters Pi can be constants, attributes or UDT methods applied to other parameters. The F function will be selected according to all the parameter types. Methods may also appear according to their complete name (as in their declaration) and thereby bypassing the inheritance based selection mechanism. The following examples demonstrate the various possibilities:

Example 1 : A UDT method appears in the projection clause. The query selects all attributes in the relation in addition to the surface value of the sides attribute.

```
SELECT    *, SURFACE (SIDES)
FROM      RECTANGLES ;
```

Example 2 : UDT methods are used in a restriction clause. This query selects those rectangles with a height greater than their width.

```
SELECT   *
FROM     RECTANGLES
WHERE    HEIGHT (SIDES) > WIDTH (SIDES) ;
```

Example 3 : A UDT method is used in a join expression. This query selects rectangles with different surface values and displays the greatest of the two values.

```
SELECT   *, BIG (R1.SIDES, R2.SIDES)
FROM     RECTANGLES AS R1, RECTANGLES AS R2
WHERE    SURFACE (R1.SIDES) <> SURFACE (R2.SIDES) ;
```

4. The System Architecture

The system which implements the ideas described in this paper is version 7 of the SABRINA relational DBMS [Gardarin87]. In this section, the general system architecture is detailed. Then, the modifications which have been introduced to manipulate UDTs will be highlighted. The architecture is a three layered architecture which spans from the external user interfaces to the disk storage system:

(1) The interface machine comprises the outer layer of the system. It is responsible for allowing different end-users to interact with the system. These processes transform queries into an internal representation called Data Manipulation Protocol (DMP). Different types of interfaces are available in the system.

(2) The assertional machine is the intermediary layer of the system. This layer is responsible for transforming the operations of relational calculus in an optimized extended relational algebra tree, for managing data security and integrity, and for managing views.

(3) The algebraic machine is the internal layer of the system. This layer performs the operations of relational algebra. To maintain performances, access methods are used, cache memory is managed and efficient join and selection algorithms are implemented. Moreover, the algebraic machine manages concurrency and reliability.

Each of these machines comprises a number of functional processors. The global architecture was

not modified by the integration of UDT. However, a new processor was added to manage UDT at the algebraic machine level. The meta-base which describes the relations, attributes and integrity constraints was extended to describe complex objects and manage domain types which are not base types. Figure 1 describes the overall system architecture. Note that several external languages are available on top of the system, including PROLOG and a specific rule language called RDL1 [Maindreville88]. Rules are interpreted at the level of the assertional machine.

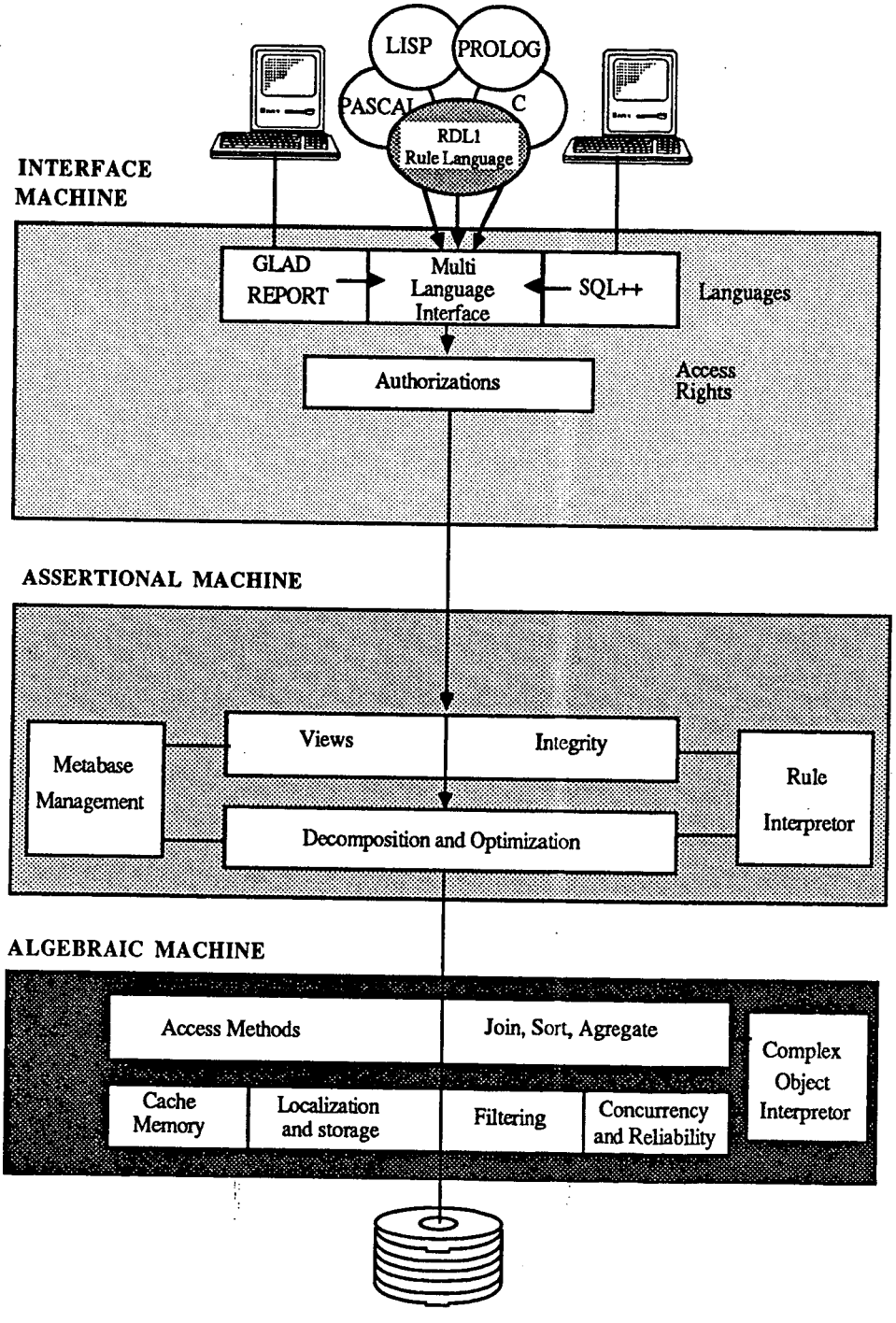

Figure 1 : Functional System Architecture.

A LISP language processor is available at the interface level to allow the definition of new UDT. All information pertaining to UDT is stored in a meta-base relation. No external files need be used with only one exception: source and object files for C code. The relation which is used to manage UDT has four attributes : i) the method name defines the name of the UDT, ii) the result type defines the domain type of result returned by the function, iii) the function type describes whether the function implements a domain or a method, iv) the function text is the source code which implements the method. A sample of this relation is given in Figure 2. A UDT processor is associated to the algebraic machine to evaluate UDT in relational expressions. This processor is called by the filter when method evaluation against relations is required. The processor is also called when clustering relations on UDT.

FUNCTION_NAME	RESULT_TYPE	FUNCTION_TYPE	FUNCTION_TEXT
#:POINT	LIST	DOMAIN	(DD #:POINT(x) (and (listp x) (numberp (car x)) (numberp (car (cdr x))) (null (cdr (cdr x)))))
#:POINT:ABCISSE	INTEGER	OPERATOR	(DE #:POINT:ABCISSE (x) (car x))

Figure 2 : A sample of the UDT relation

5. Clustering Using UDT

5.1. Principles and state of the art

A crucial problem in introducing UDT for managing complex objects in a relational system is maintaining performances at a reasonable level. While main memory storage is increasing in size, disk I/O remains the bottleneck for such systems. New applications need to manage a large number of objects where individual objects can be big in size. Applications such as geographic applications employ graphics and consume important amounts of memory and processing time. Considering the important cost of disk I/O, appropriate clustering methods are essential.

INGRES [Stonebraker86] implements several clustering methods (B-trees, hashing), and allows adding appropriate clustering methods for complex objects. Thus, it is possible to add clustering methods designed for spatial objects such as R-trees [Gutman84] and k-d-trees [Bentley75]. This approach allows the user to integrate complex and appropriate clustering for UDT; however a programming error is hard to avoid and can cause the DBMS program to terminate abnormally.

In [Valduriez86], two techniques for clustering hierarchical objects are presented. An initial approach for storing objects consists in physically clustering object and sub-object. The method favors access to entire objects over access to sub-objects. The storage model is referred to as the Direct Storage Model (DSM). A second approach called Normalized Storage Model (NSM), stores atomic objects in flat relations. The method favors access to sub-objects which can be clustered according to one or more attribute values. Accessing entire objects requires rebuilding the object with nested join operations. However, rebuilding objects can be accelerated by maintaining join indices. These join indices join tuples on predetermined attribute values. This approach is appealing in that it does not require any special clustering method and in that it is easy to implement.

These procedures do not allow to define clustering of tuples in terms of general selection criteria. They allow clustering directly on attribute values but not on the result of methods frequently applied to complex attributes. Graphical applications seldom require accessing entire objects but more often require the result of a method applied to an object. Therefore, access to such objects is functional. For example, in the relation Rectangles (R#, Color, Sides), the sides attribute is of the rectangle domain which is a UDT. Users can require accessing rectangles based on the surface value of attribute Sides. This query will result in applying the surface method against the values of the Sides attribute. It is therefore important that clustering be also functional. That is, that tuples be grouped according to the surface value of Sides. Thus, tuples must be clustered according to the most frequently used methods in queries.

5.2. Clustering using UDT methods

The SABRINA relational DBMS implements a meta-method for clustering tuples that allows describing clusters using a Predicate Tree (PT) [Gardarin84],[Valduriez84]. Tuples qualifying a same criteria are clustered in the same branch of the PT. Each level of the PT divides the relation into a set of disjoint relations. The depth traversal of the PT allows partitioning the relation into finer and finer clusters dividing the sub-relation itself into a set of disjoint relations. A leaf of the PT corresponds to the set of tuples that qualify the conjunction of criteria from the root of the PT to that leaf. A catalogue manages the relationship between a leaf of the PT and physical disk blocks. The catalogue allows logical and physical independence between the PT and corresponding disk blocks.

When dealing with UDT, clustering has to be done on tuples qualifying a method result. Selecting rectangles according to Side ='(4 5)' can be optimized by clustering directly on the attribute's values. However, if rectangles are selected with surface value equal to 12, then clustering must be done based on the result of the surface method applied to Sides.

The extended version of SABRINA allows the user to cluster tuples according to results obtained from frequently applied methods defined as LISP or C functions. An appropriate partitioning of tuples using method results reduces the number of disk blocks needed to be scanned and thus the time required to process queries. This approach doesn't require implementing specific access methods for UDT. It allows multi-dimensional clustering on simple values and on UDT method results. The same methods used in queries can be used for clustering. Furthermore, in the case of LISP written methods, errors are managed by the interpreter thereby protecting the DBMS from abnormal termination.

Each level of the PT partitions the relation according to a specific value. The clustering predicates are of the form *f (Attri) op value* where f is a function, Attri is an attribute of the relation, op is a comparison operator among {=, <>, <,<=, >, >=}. The f function must be known to the system, that is, it has been previously coded and registered with the DBMS as a complex object method.

Different possibilities to partition a relation follow. The example will be based on the Rectangles (R#, Color, Sides) relation. Figure 3 illustrates a four level partitioning of the Rectangles relation. The first level partitions the relation by applying a hashing function on the rectangle number. The second level partitions the relation by again applying a hashing function but to the result of the surface method applied to the SIDES attribute. The third level uses a ranking function to partition the relation according to a list of predicates of the form Attri = value. The forth level again uses the ranking function but at this level, the list of predicates includes UDT methods.

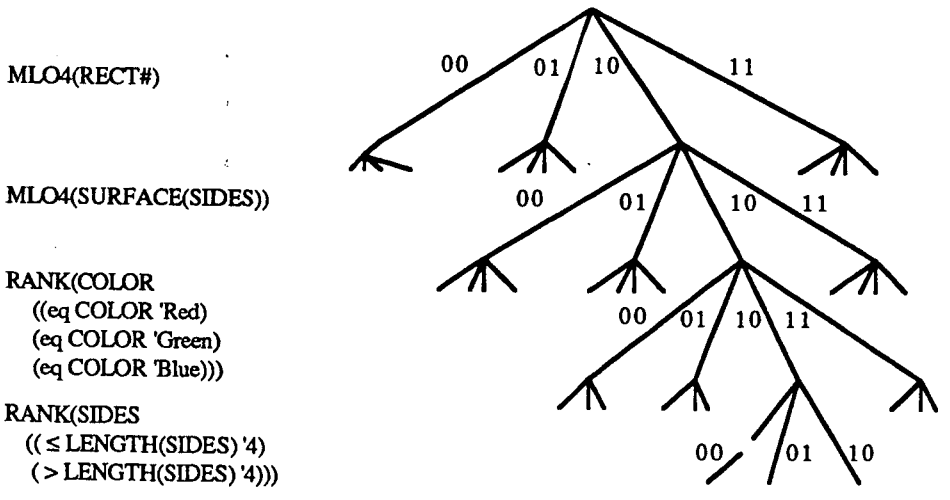

Figure 3 : Example of a Predicate Tree manipulating UDT methods

Branch 0 at level 3 constitutes the 'other' branch reserved for tuples qualifying none of the predicates in the list of predicates. The 'other' branch of level 4 is never used because the union of both predicates at this level coincide with all values included in this domain. The external language

command which is used to define the clustering of a relation is the following:

```
CLUSTER RECTANGLES ACCORDING TO
(R#                 ML04)
(surface(Sides)     ML04)
(Color              =RED =GREEN =BLUE)
(Length(Sides)      <=4 >4)
```

The PT is stored in a meta-base relation called CLUSTER.

5.3. Determining the location of a tuple

When a tuple is inserted, it is necessary to determine to which cluster belongs the tuple. For each level of the PT, the branch to which to the tuple belongs is established. At a certain level, the tuple can only belong to a single branch since predicates create disjoint sets of tuples. The branch to which a tuple belongs can be obtained with the following procedure:

i) a standard hashing function (modulo, folding, etc.) is applied to the result of a UDT method.

ii) the rank of the predicate to which the tuple qualified indicates the branch to which the tuple belongs.

The ranking procedure makes it possible to directly enumerate values or intervals. However, an 'other' branch is imperative because a tuple may not qualify any of the predicates at that level. For one level, clustering is determined according to a single attribute. The branch to which a tuple belongs is determined by successively applying each predicate to the tuple to determine the corresponding branch. When the tuple qualifies the predicate, the process is finished for that level and it resumes for the next level until the full depth of the PT is reached.

5.4. Selection Algorithm

The selection algorithm is divided into two steps. The first is an optimization that qualifies usable predicates. The second step determines branch numbers corresponding to the optimized PT. An initial simplification consists in eliminating those predicates Qij which do not refer to a clustering attribute by replacing them with the value 'true'. Then, the following rules are applied: the predicates Qij and 'true' are replaced by Qij, the predicates Qij or 'true' are replaced by 'true'. This simplification is independent of the functions used in the query or in the PT.

Each selection predicate is then considered successively. For each level of the tree, the

predicates which can be qualified are determined along with those which are contradictory with selection predicate. Determining the non-contradiction of two predicates is a complex problem. Only two predicates of the form *f (Attri) op value* will be considered. The general form of the predicates is the following:

Selection predicate: SP: f1 (Attri) op1 value1

when enumerating values or intervals, the clustering predicate has the following form:

Clustering predicate: BPi: f2 (Attri) op2 value2

If a standard hashing function is used (F is a hashing by division, interpolation or digital), then F(f2 (Attri)) defines the set of tuples at that level. If the clustering predicate is not atomic (as in the case of an interval), then the evaluation will be done on each basic predicate. For example, 10 < R# < 20 will be evaluated by R# > 10 and R# > 20.

To eliminate clusters which do not participate in the result of a query, selection predicates and clustering predicates must be comparable. For example, the query:

```
SELECT    *
FROM      RECTANGLES
WHERE     COLOR = "RED"
AND  SURFACE (SIDES) = 12 ;
```

uses level 2 and 3 of the PT in figure 2 and not level 4. *Surface (Sides) = 12* qualifies branch 00 at the second level with *ML04 (12) = 0*. The selection predicate *Color ="RED"* determines branch 01 at level 3 of the PT. However, no information on *Length(Sides)* can be drawn from *Surface (Sides) = 12*. The Length and Surface methods are independent.

At a given level in the PT, evaluating which branches qualify a selection predicate is done by applying hashing functions or by running the SelectBranch function which returns the list of possible branches where results may be found. When the equality operator is used in the selection predicate, then only one branch contains results. If the selection function is the identity function and clustering at a level is organized around a method applied to that attribute, then clusters can be qualified. The f2 function is applied to elements to which are applied the identity function. Then, clusters can be established using the function result. For example, a level clustering on the result of the method Surface (Sides) is useful in a query selecting tuples based on the Surface method applied to the Sides attribute: f1 and f2 are the Surface method and op is equality. Now consider for example that selection is done on identity as for Sides = (3 4). To use a cluster based on

Surface (Sides), the Surface method is first applied to the constant (3 4) to obtain the result. With this result, clusters may be determined. The simplified SelectBranch algorithm is given in Figure 4.

```
function SelectBranch (SP, BPi) : Ri

    {SP is the selection predicate}
    {BPi is the list of predicates at a level}
    {Ri is the result, i.e., a list of possible branches}

begin
    if f1 = identity then calculate f2 (Attri) and substitute SP ;
    if f1 <> f2 then select all branches
    else for each branch i do
        if Sp qualifies BPi then select branch i ;
end;
```

Figure 4 : The branch selection algorithm

The number of calls to the LISP interpreter by the clustering process must be limited as much as possible. The proposed algorithm reduces the number of calls when f2 is a LISP function and f1 is the identity function. All calls to the interpreter are avoided when f1 = f2. Although the clustering method is complex, little processing time is required when accessing data.

6. Optimizing the evaluation of LISP functions

6.1. Function evaluation cycles

Methods on complex objects are interpreted by a specific LISP interpreter embedded in the algebraic machine of the relational DBMS. The processing of expressions (i.e., complex objects structured as lists) by a LISP interpreter must be reconsidered for the database context where performances need to be maintained. In this section, first the main procedures involved in evaluating LISP expressions are detailed. Then modifications and simplifications brought to these procedures are presented. The purpose for these alterations is to lessen the overhead of expression

evaluation.

Complex objects and associated methods are implemented as symbolic expressions (S-expr). A symbolic expression is the single representation structure in LISP. S-expr represent both data and programs. Hence, programs and data share the same structure. There are four main procedures in the LISP evaluation cycle : READ (compiles an S-expr from an external representation into an internal one), EVAL (evaluates a compiled expression), PRINT (displays an s-expr) and GC (garbage collection). In this implementation, two procedures have been optimized and simplified to speed up evaluation, these are READ and GC.

An S-expr is compiled by the READ procedure from an external representation (where the hierarchical quality of data are represented with parenthesis) into an internal one (where actual memory addresses are managed along with a dictionary to handle symbolic data). Values, properties and functions can be attached to symbolic atoms. For example, the code which calculates the square-root of a number can be attached to the symbolic atom 'Square-root'. Numeric terms are non symbolic, they evaluate to their actual value. The set of symbols available in the environment is contained in the symbol dictionary. The aim of the dictionary is to insure symbol uniqueness and associate different types of values to a symbol. Managing the dictionary is one of the more costly activities of the interpreter. Each symbolic atom which is read by the READ procedure must be searched for in the dictionary.

6.2. Optimizations and Simplifications

In this implementation of the interpreter, values contained in tuples are considered to be non symbolic. Atoms which would normally be considered symbolic are treated as plain text. This avoid having to manage the dictionary when expressions in tuples are compiled by the READ procedure. Once the expression has been compiled, the EVAL procedure decomposes the UDT methods into simpler operations which are the basic LISP functions. The cost of evaluating an expression is directly proportional to its complexity.

Once the evaluation cycle has finished, control is transferred to the PRINT procedure which transforms S-expr from their internal representation into an external readable format. The cost of printing a structure is directly proportional to its complexity. However, simple values like real numbers, integers or character strings are returned in their actual representation so no conversion operation takes place. The PRINT procedure is the least costly of the four basic procedures.

6.3. Garbage collection

Once the result has been printed, the garbage collection procedure is run to trace unused memory fragments (in certain implementations of LISP, this process does not necessarily occur

after the print procedure). The GC process is also responsible for the important processing overhead of LISP. During the evaluation cycle, memory is consumed to store intermediate results and structures which may be no longer necessary once the evaluation cycle is terminated. The GC process consists in locating this memory and return it to the free memory store available for allocation. At the end of an evaluation cycle, what is considered as garbage are memory elements which are not referenced by the symbols in the dictionary. Information attached to dictionary elements must persist longer than one evaluation cycle. For example, if the previous cycle defined a new function, then the function code must not be garbaged at the end of the cycle. The same is true for global variables and properties. Two general algorithms are used for garbage collection. The first consists in traversing all elements referenced by symbols in the dictionary and marking those elements as valid. Then all non valid elements can be garbaged. This operation is expensive in that the entire memory must be traversed. The second operation consists in keeping a reference count for elements counting the number of references that are made to an element. A reference count of zero indicates garbage. This operation is expensive in that reference counts must be managed.

A simplification brought to the interpreter is to allow only local variables in functions and no property lists. Therefore, only function code persists between evaluation cycles. Since the interpreter is essentially a UDT method evaluator and no user environment needs to be maintained between evaluation cycles, this restriction is not cumbersome to the UDT programmer. These two restrictions greatly simplify the task of identifying garbage in this implementation. Function code is marked as non garbage and thus valid for the lifetime of the transaction. All memory consumed during a cycle can be reused during the next cycle. The memory allocation procedure has been altered to allocate old memory fragments until those fragments have been depleted. At which point, basic system calls are run to allocate new memory. Indexes provide quick access to memory elements. So garbage collection does not hinder the performances in this context.

7. End-user high-level interface

Standard relational language interfaces offer end-users the responses to their queries in tabular form. This view of data is well adapted to the relational model and to the needs of most applications. However, the flat representation of tabular forms do not capture graphical data in a significant way.

The approach used to extend the SABRINA relational DBMS does not alter the relational view of data. The results are thus naturally expresses in the form of tables. Moreover, complex data are stored in character strings where the hierarchical quality of data is displayed with parenthesis, thus directly appreciable by the end-user. Nevertheless, users should be able to enter and display data in a manner which is appropriate with their view of it. A domain is composed of instances and

methods (which can be inherited from other domains) which are user-defined. Among the set of methods applicable to an object, a user can defined methods to display and enter data in a way appropriate to human understanding.

In the relational environment, results are expressed in tabular form. Values from standard domains (integers, real numbers, character strings) are displayed in their usual text representation. UDT are displayed in text form if the user has not registered a display procedure with the system. To display UDT, the system searches for a UDT method called DISPLAY which has been registered for that UDT or one directly above it in the generalization hierarchy. If no DISPLAY method is found, the value is displayed as a character string. Otherwise, the user's procedure is used to display values. For example, geometric data such as triangles can be displayed by a special procedure. However, a general procedure to display all polygons can also be registered at this level and inherited for displaying triangles. The same strategy is applied to enter data, the system searches for a data entry procedure called ENTER defined for that domain type.

Graphical results are represented in tabular form. These are portrayed as icons. To display the contents of such a result, the particular icon to display is pointed to with the help of a mouse. Clicking the icon displays the contents of the data in graphical form. More than one object may be displayed and superimposed on the same window. Figure 3 illustrates this possibility.

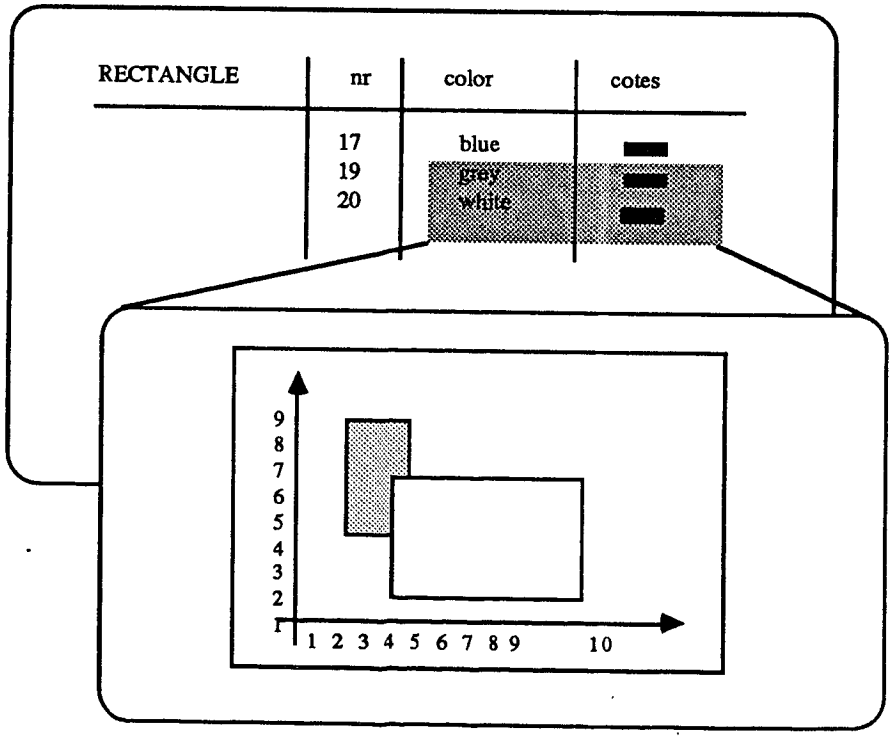

Figure 5 : Displaying graphical data

8. Conclusion

This article has described an extension to the SABRINA relational DBMS to manage User defined Data Types (UDT), which encapsulate a complex object within a set of methods. The current implementation has been described and along with the main strategies used to maintain performances. UDT are represented and programmed as LISP structures. The integrated LISP environment allows simple manipulation and implementation of UDTs. Queries can be optimized by a clustering method which allows clustering tuples according to frequently applied methods. Special methods can be defined to enter and display data in a manner appropriate to each domain type. While the interpreted environment insures protection against run-time errors in user programs, it is possible to implement UDT in C language code and dynamically link C object code to the DBMS system at run-time. C procedures are indeed called by the LISP interpreter.

To further optimize the interpretation cycle, work is being done to speed up the READ and also the PRINT phase of the evaluation cycle. At present, memory management has been simplified and the complex problem of garbage collection during method evaluation is avoided. Moreover, expressions coding complex objects in the tuples of relations are considered to be non symbolic which reduces the overhead of the system by having otherwise to manage a symbol dictionary.

In summary, UDT extends the possibilities of a relational DBMS beyond standard applications. In particular, new applications that manipulate graphical data are sought. A fully commercial version of the system is operational on SUN UNIX machines.

9. References

[AIDA87] "Aïda Reference Manual", Version 1.1, ILOG, July 1987.

[Bancilhon86] BANCILHON F, KHOSHAFIAN S. , "*A Calculus for Complex Objects*". Proc. of ACM PODS, Boston, March 1986.

[Barbedette86] BARBEDETTE G., et RICHARD P., "*VOOD: The VERSO Oriented Object Data Model*", INRIA, Research Report No. 580, Nov 86

[Bentley75] BENTLEY J.L. , "*Multidimentional Binary Trees Used for Associative Searching*" Communications of the ACM, N°9, Vol 18, 1975.

[Carey86] CAREY M. et al., "*The Architecture of the EXODUS Extensible DBMS*", Proc. of the International Workshop on Object-Oriented Database Systems, Sept 1986, Pacific Grove, California, pp.52-65.

[Chailloux86] CHAILLOUX J. , *"Le_Lisp Version 15.2 Reference Manual"*, Ed INRIA, Nov 86, 3rd edition.

[Cheiney88] CHEINEY J.P., KIERNAN G. , "*A Functional Clustering Method For Optimal Access To Complex Domaims In A Relational DBMS*", Proc. 4th Int. Conf. on Data Engineering, Los Angeles, feb. 1988.

[Codd70] CODD E.F. , "*A Relational Model for Large Shared Data Banks*", CACM, Vol 13, N°6, June 1970.

[Codd79] CODD E.F., "*Extending the Relational Model to Capture More Meaning*", ACM Transactions on Database System, Vol 4, N°4, Dec 1979.

[Gardarin84] GARDARIN G. , VALDURIEZ P. , VIEMONT Y. , *"Predicates Trees: A Way For Optimizing Relational Queries"*, Proceedings of the IEEE Computer Engineering Conference, Los Angeles, 1984.

[Gardarin87] GARDARIN G., JEAN-NOEL M., KERHERVE B., PASQUER F., PASTRE D., SIMON E., VALDURIEZ P., VIEMONT Y., VERLAINE L., *"Sabrina, a relational database system developed in a research environment"*, Technologiy and Sciences of Informatics, AFCET-Gauthier Villard-John Willey & Sons Ltd, 1987.

[Gardarin89] GARDARIN G., VALDURIEZ P. , *"Relational Databases and Knowledge Bases"*, Book, Addison Wesley, Reading, Mass., January 1989, 448 pages.

[Gutman84] GUTMAN A., "*R-Trees: A Dynamic Index Structure for Spatial Searching*" Proc. of ACM SIGMOD Conf. on Management of Data, Boston, Juin 1984.

[Kiernan87] KIERNAN G., LE MAOULT R., PASQUER F. , "*Support of Complex Domains in SABRINA DBMS: An Approach Using A Lisp Interpretor* ", 3ème Journées Bases de Données Avncées, Port-Barcares , France, May 1987.

[Maindreville88] C. de MAINDREVILLE, E. SIMON : "*Modelling Queries and Updates in Deductive Databases.*". Proc of 14th VLDB, Los Angeles, Sept. 1988.

[Ong84] ONG J. et al., "*Implementation of Data Abstraction in the Relational Data Base System INGRES*", SIGMOD, rec. 14, PP1-14, 1984

[Osborn86] OSBORN S. et HEAVEN T., "*The Design of a Relational Database System with Abstract Data Types for Domains*", ACM Transactions of Database Systems, Vol. 11, No. 3, Sept. 86, pp. 357-373.

[Schwarz86] SCHWARZ P., et al., "*Extensibility in the Starburst Database System*", Proc. of the International Workshop on Object-Oriented Database Systems, Sept. 1986, Pacific Grove, California, pp. 85-93

[Schek86] SCHEK H-J., SCHOLL M.H., *"The Relational Model with Relation-Valued Attributes"*, Information Systems, V11, N2, 1986.

[Stonebraker83] STONEBRAKER M et. al., "*Application of Abstract Data Types and Abstact Indices to CAD Data Bases*", Proc. Engineering Design Applications of ACM IEEE Database Week, San Jose, 1983.

[Stonebraker86] STONEBRAKER M., "*Inclusion of New Types in Relational Data Base Systems*", Proc. of the 2nd Conf. On Data Engineering, Los Angeles, 1986.

[Tsur84] TSUR S. et ZANIOLO C., *"An Implementation of GEM: supporting a semanticdata model on a relational back-end"*, Proc. ACM-SIGMOD Conference on Management of DATA, Boston, 1984.

[Valduriez84] VALDURIEZ P., VIEMONT Y. , *"A Multikey Hashing Schema Using Predicate Trees"*, ACM SIGMOD Conference on Management of DATA, Boston, 1984.

[Valduriez86] VALDURIEZ P, KHOSHAFIAN S, COPELAND G., *"Implementation Techniques of Complex Objects"*, Proc of the 12th Int. Conf. On VLDB, Kyoto, 1986

[Verso86] VERSO J. , "*A Database Machine based on Non 1NF relations*" reseach Report INRIA n° 523, May 1986.

[Wilms88] P.F. WILMS, P.M. SCHWARZ, H.J. SCHEK, L.M. HAAS, *"Incorporating Data Types in an Extensible Database Architecture"*, IBM Research Report N° RJ6405, Aug. 1988

[Zaniolo83] ZANIOLO C., *"The Database Language GEM""*, Proc. of ACM SIGMOD, Conf. on Management of Data, San Jose, 1983.

[Zaniolo85] ZANIOLO C ., "*The Representation and Deductive Retrieval of Complex Objects*", Proc. of the 11th Int. Conf. on VLDB, Stockholm, 1985.

Imprimé en France
par
l'Institut National de Recherche en Informatique et en Automatique